Analysing the work required in creating and maintaining a mobile application

M.Sc. in Technology Thesis
University of Turku
Department of Computing
Software Engineering
2023
Joakim Riikonen

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Joakim Riikonen: Analysing the work required in creating and maintaining a mobile application

M.Sc. in Technology Thesis, 50 p., 1 app. p.
Software Engineering
May 2023

When providing a service on the web, there is an ever increasing need for offering a mobile application as one method of usage. Developing a mobile application is expensive, however, as platform specific technologies are required when creating fully native applications, resulting in the need to build and maintain multiple separate code bases for effectively the same application.
To solve this issue, different technologies have been created that allow for multiple platforms to be targeted without having to create a separate code base for each one. Cross-platform frameworks allow for a single code base to target multiple mobile platforms, while native runtimes and progressive web application methodologies allow for web technologies to be used in the creation of a mobile application. The goal of this thesis is to evaluate these technologies in terms of work required in creating an application and the quality of the application produced. First, the benefits and drawbacks of each technology are analysed, after which two prototypes, based on the same web application, are built with the most fitting technologies. The prototypes are then compared against each other to see the differences between the technologies in practice.
For the case study performed in this thesis, the most promising technologies were cross-platform frameworks and native runtimes. The cross-platform framework React Native and the native runtime Capacitor were chosen to be used in the construction of the prototypes. The results gained from comparing the two frameworks showed that the prototype built with Capacitor was of higher quality than the one built with React Native, while also requiring less development time for its creation.

Keywords: Mobile application, React Native, Capacitor, Web application

Contents

1 Introduction

Web applications are a very popular way of offering digital services. The mobile platform is becoming increasingly more important, however, and it is common that both web and mobile versions of an application exist. Unfortunately, creating and maintaining a mobile application in addition to a web application commonly takes a lot of time and resources. This is partly due to the fact that mobile application development requires the use of platform-specific technologies, meaning that applications generally require a separate code base for each platform they target.

Different solutions have been created to avoid depending on platform-specific technologies, ranging from cross-platform frameworks to technologies such as progressive web applications, which focus on improving the web experience on mobile devices. This thesis sets out to explore these different solutions and find out how much they affect the development time required in creating and maintaining a mobile application.

1.1 Research questions and methodology

The main goal of this thesis is to review and evaluate different technologies for mobile application development, focusing on scenarios where an existing web application will serve as a basis for the mobile application. The technologies are mainly evaluated in terms of the development time required for creating an application, as well as the overall quality of the application itself, taking into account aspects like performance and user experience. This goal can be represented by the following research questions:

- **RQ1:** What challenges are there in creating a mobile application based on an existing web application?
- **RQ2:** How suitable are different mobile technologies for creating a mobile application based on an existing web application? How do they compare against each other?
- **RQ3:** How much does the chosen technology affect the amount of work required in creating and maintaining a mobile application?

In practice, the goal will be achieved by initially evaluating the technologies through a literature review. Afterwards, application prototypes will be built on two of the most promising technologies. The prototypes will be based on the same existing web application. Each prototype would implement the fundamental functionality of the application, such as authentication and translation support, and some of the core features. After building a prototype, the results are then analyzed, mainly focusing on the quality of the prototypes and the work required to implement them, as well as any further issues or otherwise noteworthy points that might have turned up during the prototype development.

1.2 Structure of the thesis

The thesis consists of a literature review and a case study. Chapter 2 goes through the background of web and mobile applications in general, while Chapter 3 goes further into different technologies that are used in mobile application development, focusing on native SDKs, cross-platform frameworks, native runtimes, and progressive web applications. Chapter 4 introduces the case study, as well as analyzes and evaluates the technologies introduced in Chapter 3 in the context of the case study. Chapter 5 goes through the process of creating the prototypes, highlighting the most significant features of the applications as well as some of the challenges that were faced during development. Chapter 6 then analyzes the results of the prototype, evaluating the quality of the final prototypes and the work required in creating them. Finally, Chapter 7 summarises the thesis and the results, going through the answers to the research questions, reviewing the limitations of the thesis, and explores options for further research.

2 Background

2.1 Web applications

Web applications are accessed by using a web browser. Just like websites, they are built using the common web technologies, such as Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript (JS). HTML is responsible for depicting the structure and content of the website, CSS describes the presentation of the web page, and JavaScript is used in scripts responsible for dynamic functionality and behaviour. [1]

HTML, CSS and JavaScript are all defined by their own standards. The HTML5 standard is maintained by WHATWG[1], the CSS standard is maintained by W3C[2], and JavaScript conforms to the ECMAScript standard, which is maintained by Ecma International[3]. Different browsers then contain implementations for the technologies that depend on the standards. Rendering engines, such as Blink[4] or WebKit[5] create a visual representation of a website based on the HTML and CSS [2], while JavaScript engines, such as SpiderMonkey[6] or V8[7], are responsible for executing the

[1]Web Hypertext Application Technology Working Group, https://html.spec.whatwg.org/

[2]World Wide Web Consortium, https://www.w3.org/Style/CSS/specs.en.html

[3]https://tc39.es/ecma262/

[4]https://www.chromium.org/blink/

[5]https://webkit.org/

[6]https://spidermonkey.dev/

[7]https://v8.dev/

JavaScript contained in a web page. Because each browser contains different implementations for the standards, differences between how web pages are displayed or what JavaScript APIs are available might exist between different browsers and between different versions of the same browser.

While there is no exact definition on what separates a web application from a website, websites are commonly considered as a collection of static web pages with the goal of providing information or content, while offering limited amounts of interactability. Web applications, on the other hand, are considered as more dynamic than websites, allowing users to manipulate the data of the application. [3][4]

Web applications can be grouped into two different categories: Multi-page applications (MPAs) and single-page applications (SPAs):

Multi-page applications consist of several different pages with mostly static content. When moving to a page or between pages, the browser requests the whole page at once, causing the browser window to refresh. As the browser requests the web page, the HTML file and the data included in it are generated on the server side, which the web browser then receives and displays.

Single-page applications consist of only one page, which contains the whole application. Because of this, the web browser does not refresh when moving between different views, as there is no need to download a new page. Requesting data from the server is done through asynchronous JavaScript requests. Instead of containing the entire page that will be displayed, the requests contain only the necessary data that the application needs, using a data format such as JSON or XML. When received by the application, the web page is then dynamically updated with the data. [5]

Because the requests contain only the required data instead of a whole HTML page, the size of the requests in single-page applications may be significantly smaller than in multi-page applications. On the other hand, because single-page applications are downloaded all at once, the initial file size may be much higher than in multi-page applications. Because of this, the usage of techniques such as lazy loading is common. With lazy loading, instead of having to download the entire application when entering a website, only the most critical resources are loaded first. Further resources are then downloaded when necessary, such as when the user navigates into a page that requires the resource in question. [6]

As web applications are accessed through a web browser, there is no need to install them like traditional applications. Additionally, many different modern devices have support for web browsers, meaning that web applications are also very accessible through multiple kinds of devices by default. However, this does not mean that accessibility requires no effort. Because the web supports such a variety of devices, numerous varying factors, such as different screen sizes and dimensions, interaction methods (mouse, keyboard, touchscreen, etc.), and browser-specific limitations have to be accounted for.

2.2 Mobile applications

Generally speaking, a mobile application is an application that runs on a mobile device, such as a phone or a tablet. Unlike web applications, which are accessed through the web, mobile applications are stored on the device and have to be installed before use. Mobile applications are usually restricted to one specific platform or operating system which they target.

Mobile operating systems are operating systems that have been designed to run on mobile devices. Currently there are two main operating systems which control the vast majority of the market share: iOS and Android. [7] Like many other operating systems, these both consist of a layered architecture, with increased amounts of abstraction on each layer. The operating systems provide a high-level API for applications to rely on. The API commonly provides features such as a view system, which allows applications to create and manage a user interface and access to system applications and hardware, such as the camera. By offering such an API, application developers do not necessarily have to consider the low-level details of the operating system when creating an application.

An example of a layered architecture is shown in Figure 2.1, which depicts the Android operating system. The highest levels contain the APIs most applications will rely on, such as the previously mentioned view system API, while the lowest level consists of the Linux kernel, which contains the drivers for the hardware of the device. The middle layers contain abstractions over the drivers inside the Linux kernel, while also encompassing native libraries such as WebKit and the Media Framework.

Mobile applications are commonly distributed through a centralized application marketplace such as Apple's App Store[8] or Google's Play Store[9]. Depending on the platform, applications might also be installable from outside of an application marketplace. This is also known as sideloading an application. For the Android operating system, applications can be freely installed from outside sources in the form of application packages (APKs), as long as the user grants permission to install unknown applications. For iOS, which is the operating system of the iPhone, installations of applications outside the App Store is essentially not possible, as Apple only allows for application sideloading during application development. It is possible to bypass this restriction by "jailbreaking" the device, but Apple strongly cautions against this and considers it as a violation of the iOS end-user software licence agreement. [9] In general this means that the distribution of applications is heavily reliant on the centralized marketplaces. This may be an issue, as all

[8]https://www.apple.com/app-store/

[9]https://play.google.com/

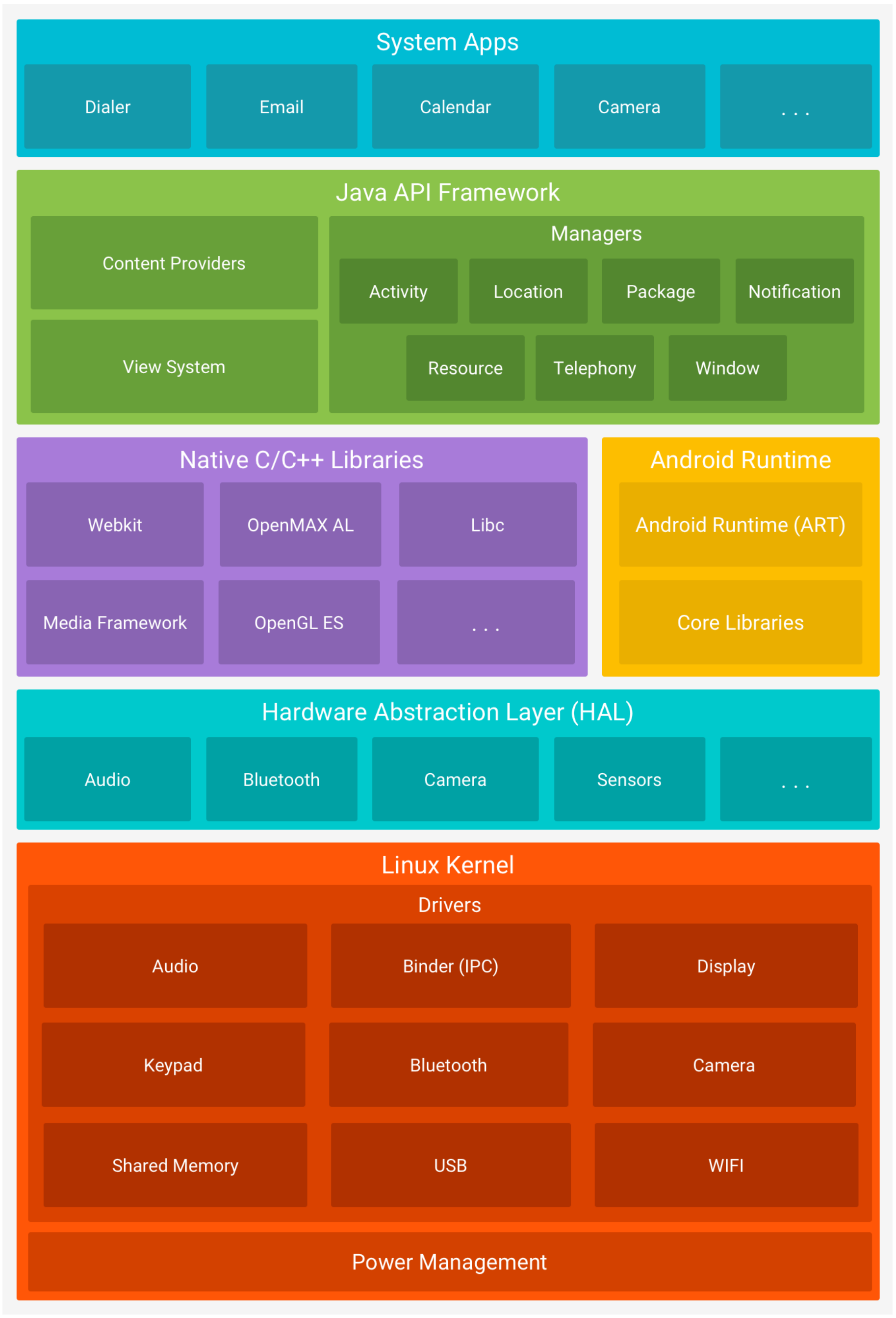


Figure 2.1: The Android software stack, from Android's official documentation [8]

applications are not eligible for a marketplace, as each marketplace has their own policies and requirements which must be abided by. In addition, the marketplaces commonly contain service fees and commissions. These include commission rates for all paid applications and in-application purchases. For both Apple and Google, the standard commission rate is 15% to 30%. [10][11] This means that if an application is distributed through a marketplace, a significant portion of all of the application's income is lost in the form of service fees.

Mobile applications can be developed with a multitude of different technologies, such as platform-specific native technologies, frameworks capable of targeting multiple platforms, and web-based technologies, where the application is run on an integrated browser on the device. These technologies will be further discussed in Chapter 3.

3 Different mobile technologies

3.1 Native SDKs and APIs

Native applications are commonly built with software development kits (SDKs), application programming interfaces (APIs), and technologies that only target one specific platform. These SDKs and APIs offer the best performance and the most support for device specific functionality, as these are fully supported by the companies behind the products themselves. The SDKs and APIs usually only offer support for specific languages. Examples of these kinds of development kits are the iOS SDK, which supports Swift and Objective-C, the Android SDK, which supports Java, Kotlin and C++, and the now discontinued [12] Windows Phone SDK.

Native applications commonly use platform-specific, pre-built UI components in constructing the UI of the application, making the user experience feel more cohesive throughout the platform. The providers of the platform in question may also offer design guidelines to make designing the UI for the native application easier. For example, Apple has created their own Human Interface Guidelines[1] for developers on Apple platforms, while the official Android documentation recommends developers to follow Google's Material Design guidelines[2] when creating their Android application.

Because the native applications are tied to a specific platform, developing an

[1]https://developer.apple.com/design/human-interface-guidelines/guidelines/overview/

[2]https://m3.material.io/

application with native languages for multiple different platforms is expensive, as multiple different code bases of the same application are required. Using a platform-specific API also makes you dependent on that specific platform, which might have negative consequences. For example, the target platform's usage might decrease or it might even get discontinued. One of the most notable platforms which this happened to was the Windows Phone, where support for the operating system ceased in 2019, therefore making all of the software written for the platform obsolete.

Common use cases for native SDKs and APIs include applications requiring high performance such as video games, applications with a large budget, where the expenses included with having multiple code bases are not an issue, and where the large cost can be justified with an improved user experience, and applications that only need to target one platform. An example would be an application that is based on a feature only available on a specific platform, or an application whose target users only use one specific platform.

3.2 Cross-platform frameworks

In order to avoid the issue of having multiple codebases for different target platforms, cross-platform frameworks capable of targeting multiple platforms while only requiring one codebase have been created. Examples of these kinds of frameworks are React Native by Meta[3], Flutter by Google[4] and Xamarin by Microsoft[5]. The performance of these frameworks, while not as good as in applications made with fully native languages, is usually good enough for most use cases, with both React Native and Flutter being capable of holding a steady 60 frames-per-second [13] [14]. Support for device functionality, such as GPS or the camera, follows in a similar

[3]https://reactnative.dev/

[4]https://flutter.dev/

[5]https://dotnet.microsoft.com/en-us/apps/xamarin

vein, where most frameworks support most features available on both iOS and Android. In case some platform functionality is not supported by the framework, or the usage of native languages and APIs is necessary, most frameworks also support integration with platform-specific languages.

Cross-platform frameworks are commonly based on a non-native language. For example, React Native is based on JavaScript, Flutter is based on Dart, and Xamarin is based on C#. This means that most frameworks either compile into a native-compatible language or offer some sort of interface between the framework language and APIs and the native language and APIs. For example, React Native offers a JavaScript layer and a native layer with a bridging layer between them, as shown in Figure 3.1, and Flutter works on top of a platform-specific embedder that coordinates with the target operating system to gain access to the required services. Xamarin, on the other hand, contains bindings between the .NET APIs and the native APIs and compiles into native ARM assembly in the case of iOS, or into an intermediate language which is then Just-In-Time compiled into native assembly during application launch in the case of Android.

One big disadvantage of using such a framework in building an application is that the application will become heavily dependent on the framework in question. This means that if, for example, the framework loses support and becomes deprecated, the application will now rely on an outdated framework and switching to a better alternative will be expensive and take time. An example of this happening to a framework would be the now discontinued MoSync framework. MoSync was a cross-platform mobile application SDK, which allowed for mobile applications to be created using HTML5 and C/C++. [16] The SDK was discontinued and lost support when the company developing it, MoSync AB, filed for bankruptcy in 2013. [17]

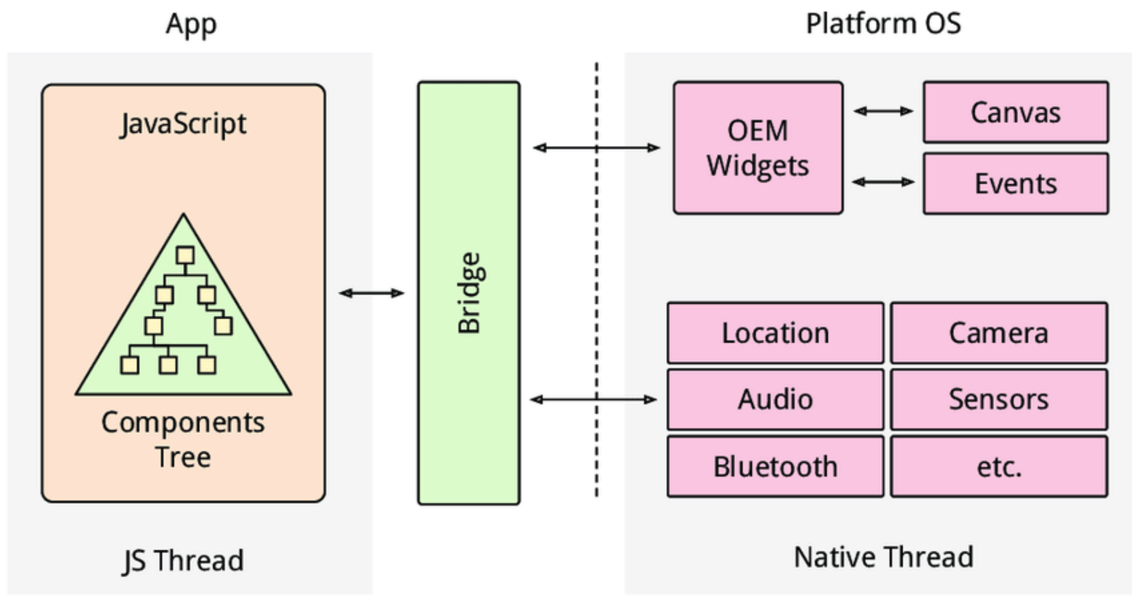


Figure 3.1: React Native application architecture [15]

3.3 Native runtimes for web applications

It is also possible to transform a single-page web application directly into a mobile application by using a native runtime such as Capacitor[6] or Cordova[7]. These types of frameworks function through the use of a WebView inside a native application wrapper. WebViews are native application components available on both iOS and Android which allow for apps to display web content as part of the app itself, usually without UI elements such as the URL field in order to make the web content feel more native to the application. Web content inside a WebView works like any other web page or web application, with completely functional HTML, CSS, JavaScript, and network capabilities. Because they are typically executed within a sandbox, applications within WebViews do not normally have access to native features.

In addition to the WebView, frameworks such as Capacitor or Cordova also pro-

[6]https://capacitorjs.com/

[7]https://cordova.apache.org/

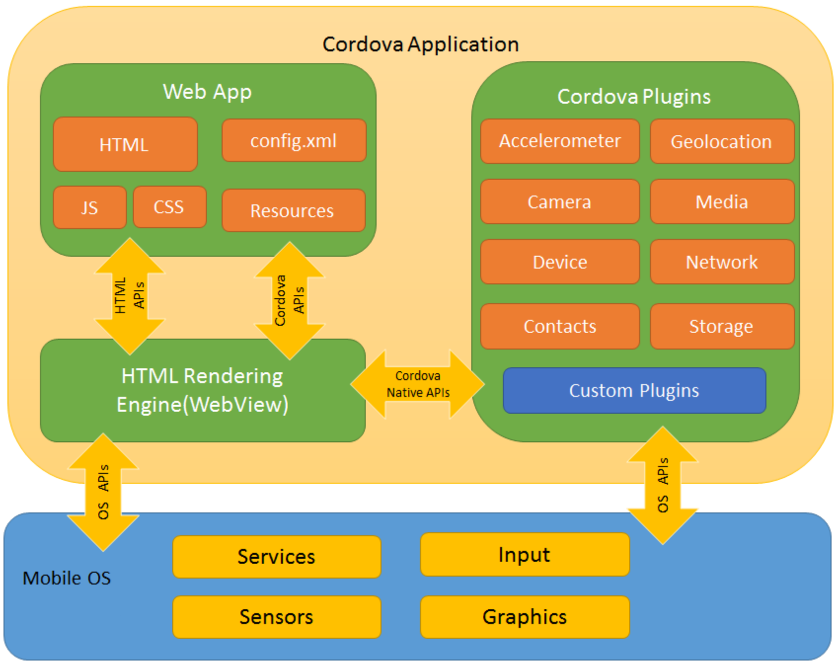


Figure 3.2: Cordova application architecture, from the official Cordova documentation [18]

vide JavaScript APIs that can be called from inside the WebView. These JavaScript APIs then invoke plugins which function as an interface between the framework and the operating system API, granting the application access to device features such as the camera or the device storage. Both Capacitor and Cordova maintain a set of official plugins, which provide access to the most commonly used APIs, but several third-party plugins maintained by the community are also available.

The relationship between the web application, the plugins, and the mobile device is further illustrated in Figure 3.2, which describes the architecture of a Cordova application.

Native runtimes for web applications offer many benefits when compared to native applications. Because a website can be transformed into an app directly from the HTML, CSS and JavaScript source, there is no need for separate code bases when bringing a web application to mobile platforms. Because of this, bringing the application to mobile is also a lot faster and cheaper when using a native runtime instead of creating a native app, as more traditional app development will cost more time and resources.

When compared to cross-platform frameworks, native runtimes also offer an advantage relating to dependency. While applications made with cross-platform frameworks are usually fully bound to the framework being used, and have to be remade completely if there is a need to change frameworks, with native runtimes, the web application stays mostly separate from the runtime itself. The native runtime functions as a wrapper around the web application, meaning that if there is a need to switch to another technology, only the code relating to the native runtime itself has to be redone.

The main disadvantages of native runtimes are related to performance and access to device features. Because the applications made with these kinds of frameworks are basically websites running inside WebViews, the performance is worse than that which could be offered by a native application. Additionally, at least in the case of Capacitor and Cordova, native device features are accessed via open-source plugins maintained by the community. This means that when developing on these platforms, the developer has to rely on these community-maintained plugins, which might wary in quality and otherwise lack behind their native counterparts.

3.4 Progressive Web Applications

Progressive Web Applications (PWAs) are web applications that can still give a native-like experience to the end user. They are accessible both on the web and as an "application". In reality, the application is just a modified browser window which displays the web application while giving it the resemblance of a native application. [19]

Progressive Web Applications offer many advantages. Because they are still fundamentally web applications, they can be indexed through search engines, increasing their discoverability. They are installable through the browser, and users can add PWAs to their mobile home screens, giving them easy access to the application. This increases their accessibility compared to normal web applications and even enables offline access to the application by downloading the site content and caching other potential data on the device. This is achieved through the use of a service worker, which essentially serves as a proxy server between the web application and the web server. [20] In addition to enabling offline access, service workers also have other responsibilities such as enabling the application to perform data synchronization in the background as well as reacting to push notifications. PWAs support progressive enhancement, where features of the application can be progressively enabled on more modern and capable browsers and devices, while still keeping the experience as good as possible on older, less capable browsers, meaning that the application offers the best possible experience for every user. [19]

Another advantage of PWAs is that the creation of a mobile "application" by adding PWA capabilities to an existing web application is much faster and cheaper than the creation of a separate mobile application. To create a basic PWA, the addition of an application icon, a secure domain (HTTPS), a service worker and a web application manifest would be enough. The web application manifest contains information about the application such as the name and description of the application, the display settings and the index URL which is shown by default when the application is started. [21]

Progressive Web Applications also have access to some native device features, such as push notifications and geolocation. Unfortunately some features may be either completely inaccessible or only accessible on some platforms. For example, at the time of writing this thesis, out of Chromium, Firefox, and Safari, only Chromium offers Bluetooth support. This is especially a problem on operating systems like iOS, which only allows browsers based on the WebKit engine to be installed, meaning that features not supported by WebKit are completely inaccessible on the entire iOS platform. Additionally, the Firefox browser has cut back on PWA support in general [22], which highlights the issues of compatibility in PWAs even more, while also making it uncertain if the support for progressive web applications will be scaled down even more in the future.

4 Analyzing the problem

4.1 Case Study

This thesis focuses on solving the problem for a specific client company with a web application that is currently available only on the web. While the web application is completely usable on mobile browsers, creating a native or native-like application that could take advantage of mobile features, such as push notifications, would be desirable.

The application is a single-page application built with the React library. In addition to React, the application also uses plenty of other JavaScript frameworks, such as Redux[1] for state management. The app has two different versions, which target different countries and contain features specific to those countries, and also supports full translation of all texts for multiple languages. Even though the app has two different versions, the versions still share the same code base, and the country-specific features are enabled or disabled through configuration files during the deployment process. The application also contains a suite of automated unit tests that target the main functionalities of the application, which are continuously maintained and updated throughout the development process.

The application is dependent on two other services: an intermediary API that serves data to the web application, and an identity provider (IDP) service that is

[1]https://redux.js.org/

responsible for authentication and authorization. The application is fully dependent on up-to-date data from the API and the IDP, so offline use of the application is not a possibility and therefore will not be a requirement of the mobile application.

Authentication to the application functions through a redirect to a login page under a separate domain where the IDP resides, where the user logs in and is then redirected back to the web application. This is noteworthy in the sense that this type of redirection could cause issues when creating the mobile application prototype, but fortunately the IDP service does also offer support for authentication through a REST API call, so using the web-based login page is not mandatory. The interactions between the web application, the API server, and the IDP service during the authentication and data fetching processes are further visualized in Figure 4.1.

In order to fully take advantage of the mobile platform, the application has some explicitly defined requirements:

- The application must be available for both iOS and Android.
- The application must be capable of being available on the App Store for iOS and on the Play Store for Android. Naturally, the prototype itself will not be available on an app store, but the app itself must be app store compatible.
- The application must support push notifications.
- Signing in to the application must be easy, but still secure. The application could take advantage of mobile security features such as biometrics.
- Although the application cannot support offline usage, it should still gracefully handle being unable to access the API or the IDP.

During the case study, two different prototypes based on the original application will be built, with two different technologies. As they are prototypes, they will not contain the full feature set of the original application. Instead, the prototypes

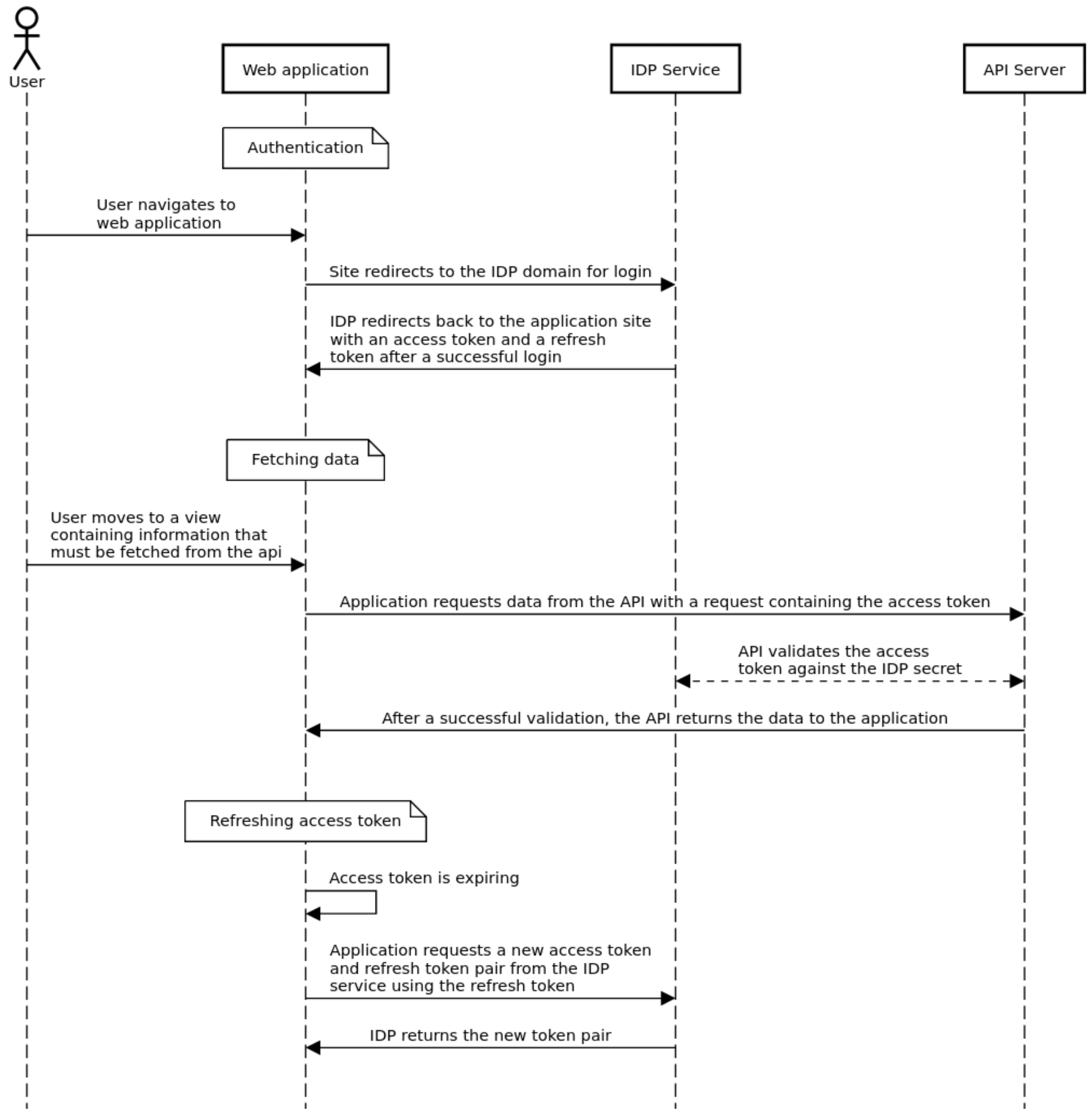

User
Web application
IDP Service
API Server
Authentication
User navigates to web application
Site redirects to the IDP domain for login
IDP redirects back to the application site with an access token and a refresh token after a successful login
Fetching data
User moves to a view containing information that must be fetched from the api
Application requests data from the API with a request containing the access token
API validates the access token against the IDP secret
After a successful validation, the API returns the data to the application
Refreshing access token
Access token is expiring
Application requests a new access token and refresh token pair from the IDP service using the refresh token
IDP returns the new token pair

Figure 4.1: Sequence diagram of the interactions between the web application, API server and IDP service during authentication and data fetching processes

will contain implementations for fundamental features such as authentication, state management, and internationalization. In addition, a set amount of basic features that are included in the original application will be implemented as well. Suitable technologies for the prototypes are analyzed in the following sections, of which the two most fitting will be chosen. The amount of work required in implementing the prototypes will be logged, and the data gathered is then later used to compare the two technologies in terms of what was achieved, how much work it took and how much work would be required for a full implementation.

4.2 Native application as a solution

Creating the application with fully native technologies would offer two main advantages: The best performance and the best access to native features possible. Unfortunately, using fully native technologies also comes with a large drawback, which is the amount of work required. At least two different applications would have to be built and eventually maintained, one for the iOS platform and one for the Android platform. By taking into account the already existing web application, this would result in having to maintain a total of three different applications, each made with different technologies targeting the different platforms. In addition to demanding the most work from the developer team, this would also end up requiring the most knowledge from the team due to the amount of different technologies used.

The main benefits gained by using fully native technologies are also not that relevant to this application specifically. The application is mainly used for viewing and submitting data, which does not require much performance-wise. Therefore, the performance gained by using a native technology would most likely have negligible impact on the end user experience. In a similar fashion, the application only demands support for notifications and, potentially, biometric authentication, as listed in the application requirements. Through official and community-maintained

APIs, packages, and plugins, these features are also available for hybrid frameworks [23][24], native runtimes [25][26], and even web applications [27][28], so it would not be necessary to create a fully native application instead of using one of the above technologies.

As a result, fully native technologies will not be used in creating the prototypes, as the negligible gains in performance would not justify the large amount of work required in building and maintaining the application.

4.3 Cross-platform framework as a solution

Compared to fully native technologies, the main advantage of using a cross-platform framework would be, as the name suggests, the ability to target multiple platforms without having to build multiple codebases. In the case of this application, it would mean that a total of two codebases would have to be maintained, one for the web application and one for the mobile applications. The disadvantages in using a cross-platform framework relate to decreased performance and lesser support for native features, but as discussed in Section 3.2, cross-platform frameworks are very much capable of performing well enough for this kind of an application. As for support for native features, namely notifications and biometric authentication, the chosen framework does not offer an out-of-the-box solution that would work on both iOS and Android. This means that the application will have to either rely on a community-maintained package or build its own abstractions around the required native features and implement them separately for both platforms, which would require a lot of work. Regardless, achieving support for the native features is still certainly achievable. Therefore, cross-platform frameworks look like a suitable technology for building the application.

For this reason, one of the prototype applications will be built with a cross-platform framework. There are multiple different technologies to choose from, such as the previously mentioned React Native, Flutter, and Xamarin, but for practical reasons the framework of choice for this prototype will be React Native. The developer team responsible for this application already has some existing experience with React Native, while having none with other cross-platform frameworks. Additionally, as the original web application has been built with React, one could assume that rebuilding the application would be easier with React Native than with another framework.

4.4 Native runtimes as a solution

Using a native runtime such as Cordova or Capacitor would allow for the existing web application to be directly converted into a mobile application by using the native wrapper provided by the framework. Building the mobile application like this would, at least in theory, greatly reduce the amount of work required. Additionally, by using a technology like this, it might be possible to create and maintain the mobile application without having to maintain more than one codebase in total, since the web application is directly converted into the mobile application. These are both very big advantages when comparing this technology to other options and, as such, make it seem like a very promising option.

Unfortunately, using a technology like this will bring with it some notable disadvantages. First of all, the performance of the application could suffer significantly, since the mobile application would basically be a web application running inside an integrated browser. The performance loss could end up being severe enough that it would start affecting the user experience of the end users. Additionally, even if the web application can be directly converted into a mobile application, the application will still require some changes specific to the mobile application. For example, the integrated browser will not contain the page history navigation buttons that a browser normally contains, so the inclusion of extra UI elements might be necessary to maintain the user experience. Naturally, the features regarding notifications and biometric authentication would be completely new and would require work as well. Finally, in both Cordova and Capacitor, native functionality is achieved through the use of plugins, as discussed in Section 3.3. Some of these plugins are official and maintained by the teams behind the frameworks themselves, while others have been created and are maintained by the community. In Capacitor's case, there is an official plugin for notifications, but only community-maintained ones for biometric authentication. This would mean that if Capacitor was to be used, the application would have to rely on a plugin maintained solely by the community.

Because a native runtime has the potential to greatly reduce the amount of work required in creating and maintaining the application, one of the prototypes will be built with a native runtime, even though it may be accompanied by some significant disadvantages. The framework that will be used in the prototype will be Capacitor, as it seems like a more modern solution than Cordova, and is even considered as a successor to Cordova by some. [29] Although it would be interesting to use both of the frameworks to see what their differences are in practice, unfortunately the time and resources available for the thesis are limited and the amount of prototypes had to be limited to a total of two.

4.5 Progressive Web App as a solution

Transforming the existing web application into a progressive web application would likely be the fastest way to create a mobile-friendly version of the application. It would have the benefit of not having to create and maintain multiple codebases, as the PWA capabilities would just be added to the existing application. These attributes make PWAs in general a very solid option for quickly bringing a web application to mobile platforms.

Unfortunately, PWAs are far from a perfect solution, as they currently have several shortcomings when it comes to distribution and native feature support. Most of these are Apple and iOS related. Firstly, on iOS, progressive web application installation is only supported on the Safari browser [30], so if the user was using another browser, such as Chrome or Firefox, installing a PWA is not possible. Secondly, web browsers on iOS lack support for some critical features, namely push notifications [27]. This means that, at the moment, it is simply not possible to send notifications through a web application on iOS, which is one of the key requirements of the application. Fortunately, this will change in the future, as Apple has mentioned that they will be adding support for Web Push for iOS and iPadOS in 2023. [31] The final major issue is that the final application must be distributable through the Apple App Store. According to multiple articles, however, PWAs are not accepted to the Apple App Store. [32][33] This sentiment is also reflected in Apple's App Store Review Guidelines, where it is declared that "Your app should include features, content, and UI that elevate it beyond a repackaged website.". [34]

For these aforementioned reasons, adding progressive web application capabilities to the existing application is unfortunately not a viable approach in this case. This is a shame, since overall PWAs seem like a very good method for enhancing the mobile experience of an existing web application, and will certainly be a good fit for some projects. With the increasing amount of support PWAs are receiving, perhaps in the future it will be a more suitable choice for projects like this as well.

4.6 Summary

Each technology has its strengths and weaknesses, with some being more relevant to this case study than others. Native applications offer the best performance and native feature support, but due to the fact that a separate code base would be required for each target platform, the amount of work that would be required makes this approach unfeasible. Cross-platform frameworks, on the other hand, can target multiple platforms with the addition of just one code base, while also offering good enough performance and support for native features. Therefore, the first prototypes will be built with the cross-platform framework React Native.

The need for additional code bases can be eliminated altogether by using solutions such as native frameworks or progressive web applications, but these bring with them their own issues. PWAs have major issues regarding feature support, which makes it unfeasible for this case study as it is unable to fulfill some of the key requirements of the project. Feature support can be an issue with native runtimes as well, but to a lesser extent. Additionally, performance may become an issue, especially on more demanding tasks. Regardless of the drawbacks, native runtimes seem like a suitable choice, so the second prototype will be built with the native runtime Capacitor.

The pros and cons of each technology are also listed in Table 4.1

Table 4.1: Summary of how suitable the different technologies are for this case study

Solution	**Pros**	**Cons**	**Used in a prototype**
Native application	Best performance Best native feature support	Requires one codebase per target platform	No
Cross-platform framework	Can target multiple mobile platforms with one codebase	Drop in performance Lesser support for native features	**Yes**
Native runtime	Shared codebase between web and mobile platforms	Drop in performance Lesser support for native features	**Yes**
Progressive Web Application	Fastest way to create a mobile-friendly version of the web application	Major issues regarding feature support	No

5 Implementation of the prototypes

5.1 Hypotheses

Regarding the amount of work required, the hypothesis of this thesis is that converting the existing web application into a mobile application with Capacitor will take less time than creating the complete mobile application with React Native. This would be due to Capacitor not requiring the creation of a completely new code base for the mobile application, instead allowing for the original code base to be used with the mobile application, thereby reducing the work required by a significant amount. It should be noted that the actual time used for both prototypes will most likely be similar. With the Capacitor prototype, the entire application will be converted, while with the React Native prototype, only the core functionality of the application, alongside a single feature, will be implemented.

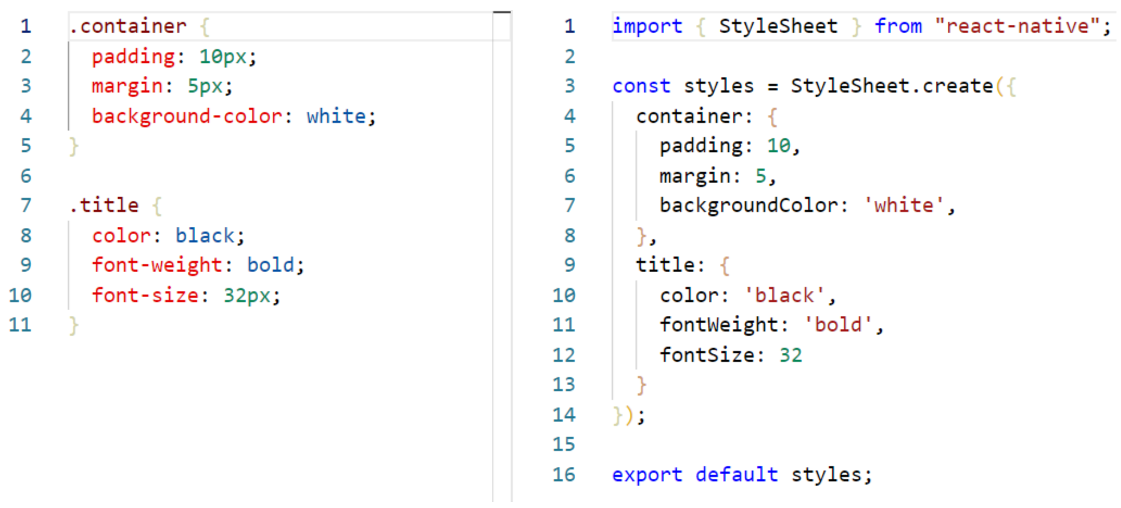
On the other hand, regarding performance, the hypothesis is that the React Native application will perform much better and will be of higher quality overall than the Capacitor application. As shown in Figure 3.1, React Native contains a native layer which is responsible for rendering the UI, among other things, while the Capacitor application is contained inside an integrated browser. Therefore it would be fair to assume that the native characteristics of React Native application would allow it to have better performance than the browser-based Capacitor application.

5.2 React Native prototype

In general, the creation of the React Native prototype was relatively smooth. This was largely due to the fact that the original web application was made with React, which meant that the overall structure of the React application code could be used as a basis for the React Native application. Because both frameworks are based on JavaScript, it was even possible to use some of the web application code in the prototype without any modifications. For example, it was possible to directly copy the state handling logic from the original React application into the React Native application. Because of these similarities, most of the actual work during the prototype development went into transforming web components into React Native components, and the authentication flow.

One of the main differences between React and React Native is the core components the frameworks use in building the UI. As React is a framework used for web development, it uses HTML-like markup called JSX to form the UI [35], which is not compatible with React Native. Instead, React Native uses Native Components. These are components that contain implementations for both the Android and iOS platforms, where the corresponding implementation is selected by React Native during runtime based on the platform the application is running on. [36]

Just like traditional web applications, React uses CSS to style the JSX components. Relatedly, React Native uses a CSS-like technology in styling components, with the main difference being that the styles are written with JavaScript instead of CSS. The styling in React Native strives to match how CSS works on the web, with many of the keywords being shared, although some differences do exist. [37] For example, instead of using pixels or other units, component dimensions in React Native are unitless, and represent density-independent pixels. In practice though, the dimensions function in largely the same way, and the pixel dimensions used in CSS can usually be directly converted into the unitless dimensions in React Native.



```css
.container {
  padding: 10px;
  margin: 5px;
  background-color: white;
}

.title {
  color: black;
  font-weight: bold;
  font-size: 32px;
}
```

```javascript
import { StyleSheet } from "react-native";

const styles = StyleSheet.create({
  container: {
    padding: 10,
    margin: 5,
    backgroundColor: 'white',
  },
  title: {
    color: 'black',
    fontWeight: 'bold',
    fontSize: 32
  }
});

export default styles;
```

Figure 5.1: Code snippet showing the similarities between CSS and React Native styling. CSS is shown on the left and React Native code is shown on the right.

A comparison of the syntax CSS and React Native uses for styling is shown in Figure 5.1.

When transforming the React components into their React Native counterparts, the main hurdles were with forms and form components. HTML contains many different types of inputs, such as buttons, text fields, checkboxes, and drop-down lists, which are all supported on the vast majority of browsers. React Native, on the other hand, lacks most of these types of ready-made inputs, containing out-of-the-box support only for buttons in the form of Buttons and Touchables[1], and text input[2]. This means that other input components, such as the checkboxes or drop-down lists, have to either be made from scratch using the available React Native components, or a third-party library has to be used. In this prototype specifically, a library was used for the checkbox implementation, while other forms of input such as the drop-down list were implemented from scratch.

[1]https://reactnative.dev/docs/handling-touches

[2]https://reactnative.dev/docs/textinput

In addition to the form inputs, HTML contains form tags which are used in encapsulating forms and enable users to submit data to a web server. The React application takes advantage of these form tags for most of its form logic. Unfortunately, React Native does not offer any direct replacement for the HTML form tags, which means that alternate approaches have to be used for creating forms in React Native. Third-party form libraries, such as Formik[3] or React Hook Form[4] may be used for implementing the missing form functionality, and many of these libraries are compatible with React as well. In this case, however, the forms in the prototype were simple enough that no third-party library was necessary. Instead, the form state was controlled through the React useState hooks, which is noted as a standard approach in the React documentation. [38]

Authentication also works quite a bit differently in the mobile prototype than in the original web application. The web application uses a library for handling the authentication flow which did not contain support for React Native, so the authentication flow had to rebuilt completely.

As described in Section 4.1, the web application authenticates the user by redirecting them from the web application domain to the IDP domain for login. In the prototype, the redirection works the same as in the web application, except instead of redirecting from a web domain into another web domain, the user is redirected from the application to the IDP domain using an integrated browser, and then the IDP site redirects the user back to the mobile app by using a deep link[5] on Android or a universal link[6] on iOS. It should be noted that deep links are not secure and should not contain sensitive data. The deep link scheme can be arbitrarily chosen by the developer, meaning that hijacking a deep link for an application is possible by

[3]https://formik.org/

[4]https://react-hook-form.com/

[5]https://developer.android.com/training/app-links/deep-linking

[6]https://developer.apple.com/ios/universal-links

simply creating another application with the same deep link scheme. This security flaw can be and is being mitigated by using a authorization flow such as PKCE[7], where the client application generates a secret which is used when requesting the access token after authentication, blocking potential hijackers from gaining access to the access token.

Then, instead of using a web-only technology such as cookies or localStorage to store the authentication tokens granted by the IDP, they are stored inside an encrypted container on the device. The implementations for the encrypted container is known as KeyStore[8] on Android and KeyChain[9] on iOS, but in order to avoid the need for platform-specific code, the library react-native-keychain[10] was used as an abstraction over the platform-specific credential stores.

As demonstrated by Figure 4.1, the web application uses access tokens and refresh tokens to authenticate requests to the API. The access token has a lifespan of 2 hours, while the refresh token used to request a new access token has a lifespan of 4 hours. Because we do not want the user to have to log in every time they open the mobile application, this 4 hour lifespan is not enough. Therefore, instead of using a refresh token with a lifespan of 4 hours to request the new access tokens, we use an offline token, which functions otherwise just like a refresh token, except it has a lifespan of 30 days. Just like the refresh token, this token is also refreshed every time the access token is refreshed. This means that the user will only have to log in when they have not used the application for over 30 days.

The authentication flow of the initial user authentication is pictured in Figure 5.2. Fetching data and refreshing the access token still function similarly in the prototype as in the web application, as depicted in Figure 4.1.

[7] https://oauth.net/2/pkce/

[8] https://developer.android.com/training/articles/keystore

[9] https://developer.apple.com/documentation/security/keychain_services

[10] https://github.com/oblador/react-native-keychain

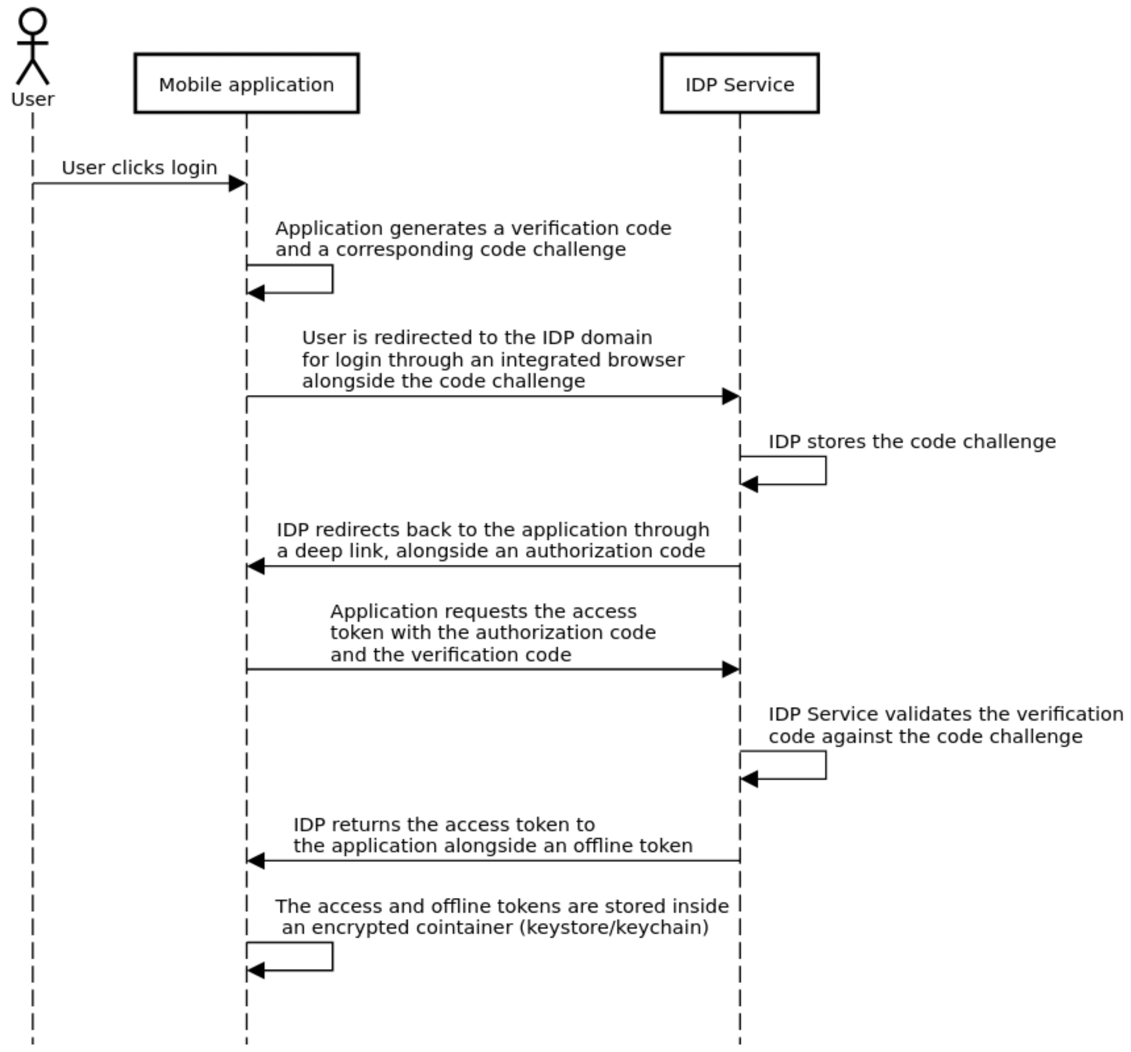

User
Mobile application
IDP Service
User clicks login
Application generates a verification code and a corresponding code challenge
User is redirected to the IDP domain for login through an integrated browser alongside the code challenge
IDP stores the code challenge
IDP redirects back to the application through a deep link, alongside an authorization code
Application requests the access token with the authorization code and the verification code
IDP Service validates the verification code against the code challenge
IDP returns the access token to the application alongside an offline token
The access and offline tokens are stored inside an encrypted cointainer (keystore/keychain)

Figure 5.2: Sequence diagram of the interactions between the mobile application and the IDP service during the initial user authentication

Finally, a couple things should be noted about development and testing. The prototype application was developed on the Windows operating system, and tested on an emulator as well as a real Android device through USB debugging. The application testing was unfortunately limited only to Android. The company development environment was only accessible through in-house computers, which were only available with the Windows operating system, and as building the application for an iOS device requires an Apple Mac, testing on iOS was not a possibility. While the application was not tested on iOS, it should theoretically still function on the operating system with only some additional work required, as platform-specific code was intentionally kept at a minimum, with only unavoidable platform-specific code, such as application configuration, being created as necessary.

5.3 Capacitor prototype

Transforming the web application into a Capacitor application was overall pretty straight-forward. Most of the work required for creating this prototype went into creating the Capacitor configuration, modifying some of the React configuration into a form compatible with Capacitor, as well as readjusting the authentication flow to function in the context of the Capacitor application. Most of the Capacitor specific configuration is contained in a separate file and is shared for both iOS and Android, meaning that the Capacitor-specific code stays mostly separate from the web application code.

The authentication flow used in this prototype highly resembles the one used in the React Native prototype, although there are some significant differences. The authentication flow begins by redirecting the user to the IDP domain for login. In the prototype this is done by opening the login site in the default browser app of the device, which is not ideal as it adds unnecessary friction to the authentication flow. In the full implementation, this would be done by using an in-app browser, which can be accomplished by using the Capacitor Browser API[11]. After login, the user is redirected back by using a deep link or a universal link depending on the platform, just like in the React Native prototype. To handle the deep link redirect, the Capacitor App API[12] is used to respond to the event fired by the redirect, which then redirects the user to the correct page on the web application. As in the React Native prototype, the inherent security flaws contained in deep links are mitigated by using the PKCE authorization flow.

The authentication tokens acquired during the authorization process are stored inside cookies, just like in the web application. This is not ideal, as the cookies are not encrypted, and in the full implementation they would be stored inside an encrypted container, such as KeyChain or KeyStore. The reason why the tokens are not stored in the secure container is that the original web application uses a framework for for handling most of the authentication flow. The framework was clearly designed for a web-only environment and contained a very limited amount of customization options, which meant that the way the tokens are stored could not be reconfigured.

Other issues with the framework also emerged. For example, the silent renewal of the authorization token did not function, as the framework relies on using an iframe to execute the silent renewal, which does not function the same way on a Capacitor application as it does on the web. The way to solve this issue would be to either use or create a library that would work on both web and Capacitor environments. Migrating to such a library for a prototype was deemed to take too much work, which is why the authentication flow in the prototype ended up being lacking in some ways.

Apart form the issues mentioned above, transforming the web application into

[11]https://capacitorjs.com/docs/apis/browser

[12]https://capacitorjs.com/docs/apis/app

a Capacitor application was quite smooth and resulted in a fully functional mobile application. Regarding the development environment, this prototype had the same Windows-related limitations that the React Native prototype had, meaning that the application was only able to be tested on an Android emulator and an Android device. Regardless of this, with the addition of some iOS-specific configuration, the application should still work on iOS without requiring any extensive work.

6 Result analysis

6.1 Quality

The original hypothesis that was formed before making the prototypes was that the React Native prototype would be of higher quality than the Capacitor prototype due to it being based on native technologies. In practice, this would mean that the React Native prototype would perform better, would contain a better user experience, and would contain less defects or bugs than the Capacitor prototype. The quality of the applications was measured by testing their performance in terms of frames per second and CPU usage, measuring their file size, as well as testing them manually for bugs or defects.

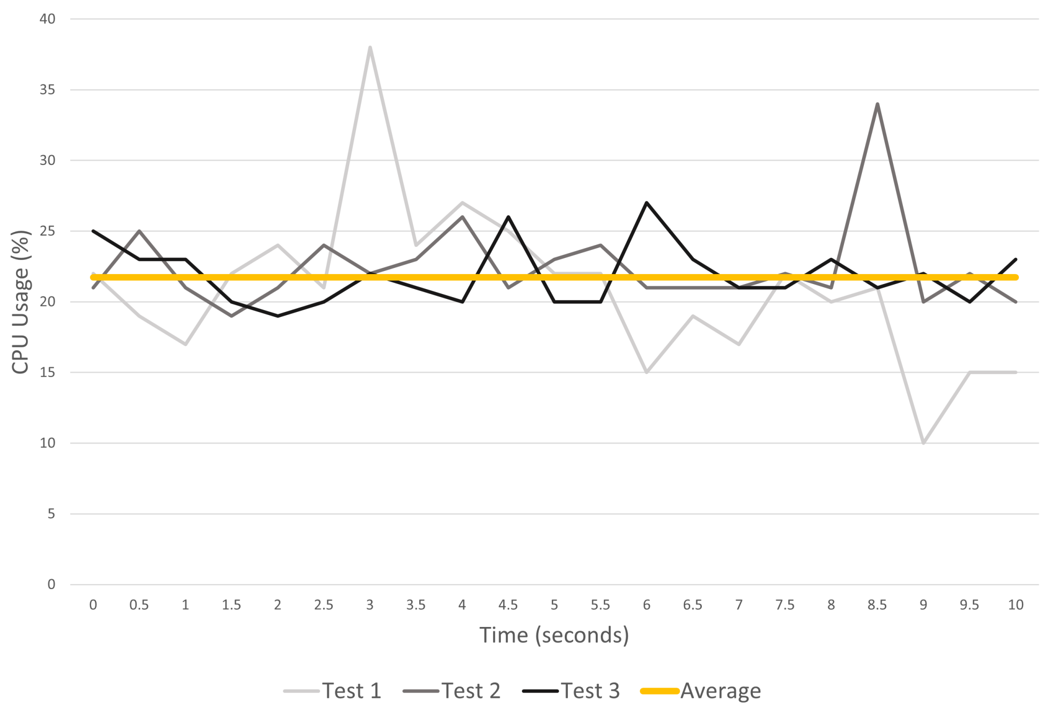
Regarding performance, neither one of the prototypes had any major issues. Both prototypes ran at a stable 60 frames per second on both the emulator and the real device regardless of the activity that was being performed on the application. In addition to measuring the frame rate of the prototypes, the CPU usage of the applications was also tested.

For the performance test, the same task is performed on both applications, and the CPU usage during the task is recorded. The task chosen for the test was scrolling through a case list view. The case list view involves loading and rendering hundreds of case card components, where each card contains multiple text components as well as an image component, making it one of the most computationally intensive tasks the prototypes contain. The tests were performed on a real device, the details of which are included in Appendix A, and the CPU activity on the device was inspected by using the CPU Profiler provided in Android Studio[1]. The results of the performance tests are shown in Figure 6.1.

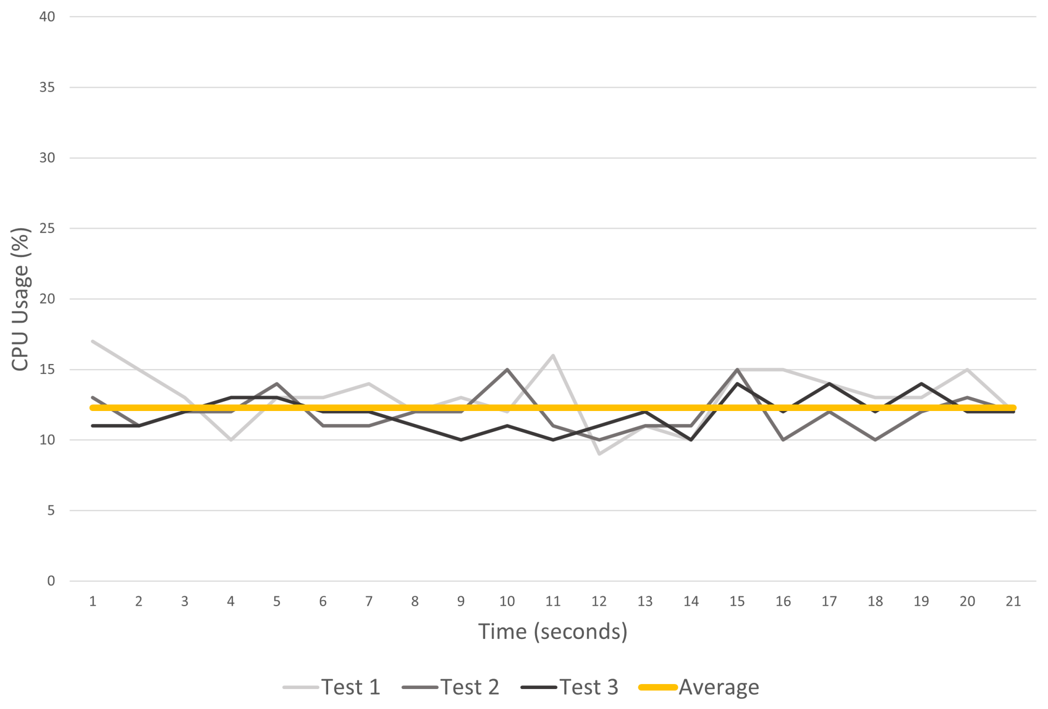
Based on the performance test, the Capacitor prototype performs significantly better than the React Native prototype. While scrolling through the list, the average CPU usage of the React Native application was 21.7%, while for the Capacitor application it was only 12.3%. This quite clearly goes against the original hypothesis, where it was assumed that the React Native prototype would perform better than the Capacitor prototype due to Capacitor using an integrated browser instead of the native components of the platform in question, like React Native does. While the difference in performance could be, at least partially, due to poorly optimized code being used in the React Native prototype, the performance difference is likely caused by the frameworks themselves.

The differences in performance between React Native and Capacitor have been observed before. In an article written by the Ionic team comparing the performance between the frameworks on iOS [39], a similar test was performed, where the CPU usage of applications was recorded while scrolling through an example list of employees. The results obtained from their test show an even higher difference in performance: The highest recorded CPU usage of the React Native application in question was "nearly 200%", whereas for the Capacitor application it was only "upwards of 10%". The article mentions that the reason for this is in the JavaScript engines being used: On iOS, Capacitor uses the much faster WKWebView engine, while React Native uses the slower JavaScriptCore engine. It should be noted that the article was written in March 2022. Since then, in September 2022, React Native 0.70 was released, which switched the default JavaScript engine from JavaScriptCore

[1]https://developer.android.com/studio/profile/cpu-profiler



(a) CPU usage of the React Native application



(b) CPU usage of the Capacitor application

Figure 6.1: CPU usage of the React native and Capacitor applications while scrolling down a list of cases in terms of percentage of the phone's CPU used. Three separate tests and an average of the tests are shown.

to Hermes[2], a JavaScript engine optimized specifically for React Native, meaning that the findings of the article are at least partially outdated. [40] Another thing to note is that the article was written by Ionic, which is the team behind Capacitor, so it is in their interest for them to portray Capacitor in a good light. Regardless of the switch in JavaScript engines, it seems that Capacitor still performs better than React Native, at least in this case.

In addition to performing better, the file size of the Capacitor prototype is also considerably smaller than the file size of the React Native prototype. The install size of the release build of the Capacitor application is 19.34 MB, while the React Native application is more than double than that with a size of 44.81 MB. While already much larger than the Capacitor application, it should also be considered that the React Native prototype contains a partial implementation, while the Capacitor prototype already contains the full implementation of the original web application. Because of this, it would be expected that the size of the final application built with React Native would be even larger than the 44.81 MB. Regardless, even though the difference between the file sizes is quite large, the React Native prototype is of an acceptable size for a modern mobile application, and is far from the 150 MB limit enforced by Google for Android App Bundles published in the Google Play Store [41].

In regards to defects, no bugs were discovered in either one of the prototypes during the manual testing of the final prototypes. This is not surprising, as the prototypes have not been extensively tested in production environments, and bugs found during development were also fixed as they were encountered. Therefore it could be said that the prototype application are of equal quality in this aspect.

Finally, while performance and defects are easy to measure and compare quantitatively, user experience is harder to assess in such a way. Fortunately, since both

[2]https://hermesengine.dev/

of the prototypes mimic the web application as closely as possible, the end result is that the user experience in both applications is nearly identical. The only major difference is in the authentication flow, where the Capacitor application redirects the user to the authentication domain through an external browser app instead of an integrated browser, as mentioned in Section 5.3. However, this is the case only due to the amount of work restructuring the authentication flow would require, and in the full implementation the Capacitor application would use an integrated browser just like the React Native application. Therefore the user experience in the application prototypes could be considered equivalent regardless of the differences in authentication flow.

In summary, the Capacitor prototype performed significantly better than the React Native prototype, while also having a smaller file size. There were no differences in the user experiences of the prototypes, and no major defects were found in either one. Therefore it would be fair to say that the Capacitor prototype was of higher quality overall than the React Native prototype, which makes the original hypothesis regarding quality incorrect.

6.2 Amount of work

Regarding the amount of work required for the prototypes, the original hypothesis was that the Capacitor prototype would require less work than the React Native prototype due to it not requiring the construction of an entirely new code base. During the development of the prototypes, the work that went into the two prototypes was measured in terms of time used. The work measured was divided into different categories based on what was being developed. These categories include initialization and setup, general development, authentication, and feature-specific categories. The measured work contains only the time that was spent actively developing the prototypes, so time spent on activities such as meetings or performance tests are not included in the results. The breakdown of the work performed during the creation of the prototypes is shown in Figure 6.2.

The initialization and setup category includes time spent on setting up the development environments as well as the prototype projects. The time spent setting up was about the same for both prototypes, and in both cases the process was relatively smooth with no major issues, since both React Native and Capacitor offer a solid "getting started" documentation that made initial setup very easy.

The general development category functions as a catch-all category for activities that did not really fall under any of the other labels. For the React Native prototype, this encompasses things like the development of generic, common components, setting up the API client, and replicating the state management logic from the original web application. In the case of the Capacitor prototype, this category mainly contains changes to the application configuration. The React Native prototype required much more of general development than the Capacitor prototype. This was to be expected, since in Capacitor the web application is directly used in the mobile application, meaning that all of the functionality already exists, whereas in the React Native application all of the functionality has to be reimplemented.

Implementing authentication took a similar amount of time for both prototypes, but for different reasons. In the React Native prototype, the authentication flow was redone from scratch to function in a mobile environment. Since the React Native prototype only targets mobile platforms and does not have to function on the web environment, the authentication flow was implemented accordingly. The Capacitor prototype, on the other hand, targets both web and mobile platforms and contains only one shared code base for all of the targeted platforms. Because of this, the existing authentication implementation on the original web application was modified to work on both web and mobile platforms. As discussed in Section 5.3, the prototype implementation of the authentication flow is incomplete and has flaws concerning both features and security, which are mostly due to the fact that the original web application uses a web-only authentication framework to handle most of the authentication implementation.

Because of this, more work would be required to create a complete authentication implementation for the final Capacitor application. To make an authentication system that both works and is secure on all platforms, both web and mobile, would take a considerable amount of research, work, and testing, which is why it was omitted from the prototype implementation. A conservative estimate would be that at least 40 man-hours would be needed to complete the final implementation. This would mean that the amount of work required to implement the authentication in the Capacitor application is much larger than in the React Native application. This should come as no surprise though, as, again, the Capacitor application has to function in more environments than the React Native application, making the authentication implementation more complex.

For the React Native prototype, although the existing React code could still be used as a guide or a template, and some of the original code could even be replicated directly into the prototype, since the application required a completely new code base, all of the features of the original web application also had to be rebuilt. In the prototype these features were limited to implementing a part of the form components required by the application, as well as the entirety of the case list feature. Implementing just these features took a considerable amount of work, and considering that the full application contains a much greater amount features than the prototype, some of which are very complex, it is evident that a major amount of work would be required to complete the application.

The Capacitor prototype, contrarily, uses the existing web application as a basis for the mobile application, meaning that there is no need to reimplement any of the features that already exist in the web application. Therefore, when compared to the

React Native application, a substantial amount of work is eliminated by avoiding the need to reimplement the already existing features.

Finally, it should be considered how the amount of work required would be affected if, in the future, both the web and mobile application are to be extended with new features. In the case where the mobile application was made with React Native, two separate code bases would exist, one for the web platform and one for the mobile platforms. This would mean that the feature would have to be implemented twice, so that both code bases would contain the implementation. In the case where the mobile application was made with Capacitor, only one code base would exist that would be shared between web and mobile platforms. Therefore only one implementation of the feature would be necessary, resulting in a smaller amount of work required. The feature still has to be implemented in such a way that it is compatible with both web and mobile platforms, meaning that it would still require more work than a web-only or a mobile-only implementation, but it would be reasonable to assume that the total amount of work would still be less than that required in creating two separate implementations.

To summarize, the Capacitor prototype required much less work in total than the React Native prototype, even when accounting for the additional work required to revise the inadequate authentication flow. This is mainly due to the fact that when using Capacitor, the features of the original application do not have to be reimplemented, as is the case with React Native. The same would hold true to changes that would come in the future as well, due to the fact that with Capacitor, the code base is shared between the mobile and web applications, avoiding the need to create duplicate implementations of shared features, as would have to be done if React Native was to be used. Therefore the original hypothesis regarding the amount of work required was correct.

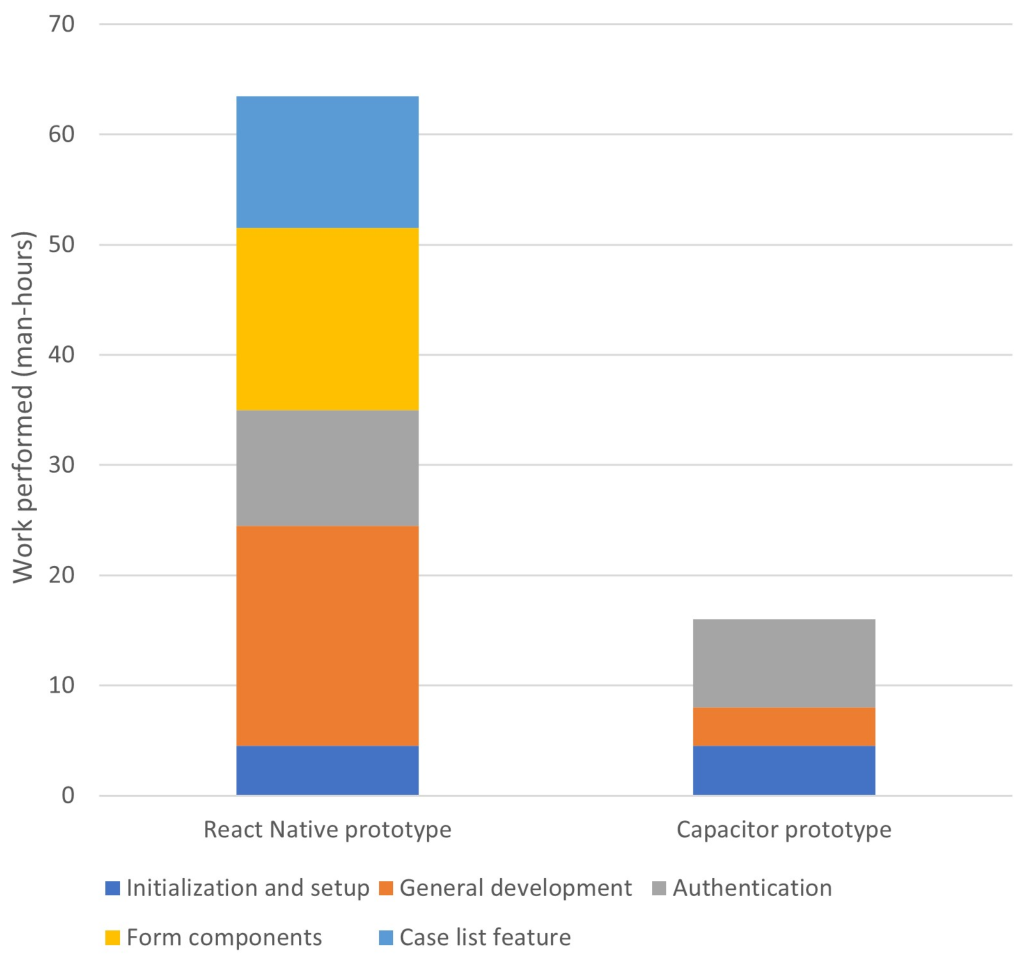


Figure 6.2: Breakdown of the work performed in the creation of the prototypes

7 Conclusion

7.1 Answering the research questions

The answer to **RQ1** (What challenges are there in creating a mobile application based on an existing web application?) was mostly discussed in Chapters 2 and 3. The main challenges encountered mainly rely on the fact that the web and mobile platforms are fundamentally different and rely on completely different technologies. Therefore, converting a web application into a native mobile application usually requires an entirely new code base built with technologies specific to the mobile platform, where all of the features of the original application have to be reimplemented, ultimately resulting in a lot of knowledge and work being required for the creation of the mobile application. Depending on the use case, browser-based mobile applications, such as native runtimes or progressive web applications, might also be an option, which may lessen the workload in creating the application, but also might require compromises relating to performance, access to platform features, and availability.

To answer **RQ2** (How suitable are different mobile technologies for creating a mobile application based on an existing web application? How do they compare against each other?), four different technologies were evaluated during Chapters 3 and 4, each of them having unique benefits and drawbacks. Native applications have the best performance and support for device features, but also require the most work and knowledge, since every target platform requires their own code base that is based on platform-specific technologies. For example, if the goal was to target web, Android, and iOS platforms, three separate code bases would be required. Cross-platform frameworks allow for the application to target multiple mobile platforms while requiring only one code base, but still requires separate code bases between web and mobile. This means that when expanding from web to mobile platforms, a new code base still has to be created, which is better than having to create multiple code bases but still requires a considerable amount of work. The drawback with using cross-platform frameworks is lesser performance and and support for device features, as well as having to bind the application to a specific framework.

Native runtimes allow for the web application to be directly transformed into a mobile application, which allows sharing the application code base between web and mobile, thus avoiding the need to create separate code bases, which ultimately results in less work required. Like cross-platform frameworks, native runtimes also compromise on performance and support for device features, while also partially binding the application to the framework being used. The binding is not as heavy as with cross-platform frameworks, though, as the web application is still clearly separate from the native runtime, meaning that if migrating to another framework is necessary, only the code relying on the framework itself has to be migrated. Finally, bringing an existing web application to mobile by transforming it into a progressive web application requires much less work than creating a separate mobile application. Using progressive web applications has some major drawbacks with having a lack of support on some platforms and browsers, as well as having no access to some device features, making it unfeasible in some use cases.

The cross-platform framework React Native and the native runtime Capacitor were more closely compared during the creation of the prototypes in Chapters 5 and 6. The results show that, at least based on the prototypes, Capacitor seemed like a better solution overall, resulting in less work required, better overall performance, and otherwise equivalent quality when comparing the prototype applications.

To summarize, there is no one definitive answer to which technology one should pick when bringing their web application to mobile, as it is heavily dependent on the use case. For example, for applications that benefit a lot from great performance, fully native applications as a good option. On the other hand, for simple applications that do not require many device features, progressive web application seem like a suitable solution. In the case of the case study performed in this thesis, the best technology options for the mobile application were React Native and Capacitor, out of which Capacitor seems like a better solution overall. In order to choose a good technology for a mobile application, the requirements of the application must be analyzed and the decision must be based on which technology fits the requirements best.

The answers to RQ2 and RQ3 have some overlap, but to answer **RQ3** (How much does the chosen technology affect the amount of work required in creating and maintaining a mobile application?) specifically, the choice of technology has a major effect on the amount of work the creation and maintenance of the application will require. Making fully native applications demands a lot of work, especially if the goal is to target multiple different platforms. Cross-platform frameworks are better in this regard, as they allow for multiple mobile platforms to be targeted without having to create multiple code bases, but will still require at least two code bases if we wish to target both web and mobile. Finally, by taking advantage of an existing web application with technologies such as native runtimes and progressive web applications, the amount of work required in creating and maintaining the mobile application is vastly reduced.

7.2 Limitations of the thesis

The main limitation of this thesis is that only two prototypes were built during the prototyping phase. Because the prototypes were built using only React Native and Capacitor, only those two technologies can be directly compared, and more prototypes would be necessary for more in-depth comparisons between all technologies. The creation of a prototype with at least one native technology would have served as a solid baseline for comparisons with other technologies, and had multiple native prototypes been constructed for different platforms, the differences between platforms could have also been analyzed.

Different cross-platform frameworks would have also been good to compare in practice. Frameworks like Flutter and Xamarin differ a lot from React Native, and so the quality of the prototypes and the work required in creating them could have varied significantly between them. The effect of the technologies chosen for the original web application and the mobile prototype could have also been analyzed had more prototypes been created. In the case study, the existing web application was made with React and the cross-platform framework chosen for one of the prototypes was React Native. Since React Native is based on React, this likely resulted in an easier transition than if transitioning from React to Flutter, for example.

Regarding the web application-based mobile technologies, Capacitor was chosen as the native runtime for one of the prototypes due to being more modern than Cordova, but the differences between the two frameworks would have been beneficial to see in practice. The practical evaluation of progressive web applications in the context of bringing an application to mobile platforms would have also been useful, but unfortunately the technology was not a good fit for the use case of the case study.

Finally, one overall problem the prototypes had was that they were developed solely by a single developer, the author of this thesis. Developers may have varying experiences with different technologies, resulting in differences in the work required to create applications as well as the quality of the applications produced. To properly gauge the differences between the technologies, the experiences of multiple different developers would be required. Additionally, developers of multiple different skill levels should be included, as the knowledge and experience the developers have with the technologies will highly influence the end result.

References

[1] "HTML: HyperText Markup Language". (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML. (accessed 02.03.2023).

[2] "Rendering engine". (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Rendering_engine. (accessed 02.03.2023).

[3] V. Belsky, "Web application vs. website: finally answered", 2017. [Online]. Available: https://www.scnsoft.com/blog/web-application-vs-website-finally-answered.

[4] E. Designs, "Website vs Web App: What's the Difference?", 2019. [Online]. Available: https://medium.com/@essentialdesign/website-vs-web-app-whats-the-difference-e499b18b60b4.

[5] V. Solovei, O. Olshevska, and Y. Bortsova, "The difference between developing single page application and traditional web applications based on mechatronics Robot Laboratory ONAFT application", *Automation technological and business - processes*, vol. 10, no. 1, Apr. 2018. DOI: https://doi.org/10.15673/atbp.v10i1.874.

[6] "Lazy loading". (2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading. (accessed: 02.02.2023).

[7] “Mobile Operating System Market Share Worldwide”. (2022), [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/2022. (accessed: 03.02.2023).

[8] “Platform Architecture”. (2023), [Online]. Available: https://developer.android.com/guide/platform. (accessed: 03.02.2023).

[9] “Unauthorized modification of iOS can cause security vulnerabilities, instability, shortened battery life, and other issues”. (2018), [Online]. Available: https://support.apple.com/en-gb/HT201954. (accessed: 02.02.2023).

[10] “App Store Small Business Program - Apple Developer”. (2023), [Online]. Available: https://developer.apple.com/app-store/small-business-program/. (accessed 29.04.2023).

[11] “Service fees - Play Console Help”. (2023), [Online]. Available: https://support.google.com/googleplay/android-developer/answer/112622. (accessed 29.04.2023).

[12] “Windows 10 Mobile End of Support: FAQ”. (2023), [Online]. Available: https://support.microsoft.com/en-us/windows/windows-10-mobile-end-of-support-faq-8c2dd1cf-a571-00f0-0881-bb83926d05c5. (accessed: 06.01.2023).

[13] N. Hansson and T. Vidhall, “Effects on performance and usability for cross-platform application development using React Native”, M.S. thesis, Linköping University, Human-Centered systems, 2016, p. 92. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:946127/fulltext01.pdf.

[14] T. Tran, “Flutter Native Performance and Expressive UI/UX”, Metropolia University of Applied Sciences, 2020. [Online]. Available: https://urn.fi/URN:NBN:fi:amk-202005067530.

[15] P. Barsocchi, M. Girolami, and D. La Rosa, “Detecting Proximity with Bluetoot Low Energy Beacons for Cultural Heritage”, *Sensors*, vol. 21, no. 21, 2021. DOI: https://doi.org/10.3390/s21217089.

[16] “Create iPhone and Android apps with JavaScript and C++ | cross-platform mobile application development”. (2012), [Online]. Available: https://web.archive.org/web/20120428171204/http://www.mosync.com/.

[17] “Mosync AB Bankruptcy”. (2016), [Online]. Available: https://web.archive.org/web/20160507095350/http://www.allabolag.se/5566727045/verksamhet.

[18] “Architectural overview of Cordova platform”. (2021), [Online]. Available: https://cordova.apache.org/docs/en/10.x/guide/overview/. (accessed: 03.02.2023).

[19] “Progressive web apps (PWAs)”. (2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps. (accessed: 10.01.2023).

[20] “Service Worker API”. (2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API. (accessed: 10.01.2023).

[21] “Web app manifests”. (2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Manifest. (accessed: 10.01.2023).

[22] J. Newman, “Firefox just walked away from a key piece of the open web”, 2021. [Online]. Available: https://www.fastcompany.com/90597411/mozilla-firefox-no-ssb-pwa-support.

[23] “react-native-notifications - npm”. (2023), [Online]. Available: https://www.npmjs.com/package/react-native-notifications. (accessed 09.02.2023).

[24] “react-native-biometrics - npm”. (2023), [Online]. Available: https://www.npmjs.com/package/react-native-biometrics. (accessed 09.02.2023).

[25] "Local Notifications Capacitor Plugin API | Capacitor Documentation". (2023), [Online]. Available: https://capacitorjs.com/docs/apis/local-notifications. (accessed 09.02.2023).

[26] "capacitor-native-biometric - npm". (2023), [Online]. Available: https://www.npmjs.com/package/capacitor-native-biometric. (accessed 09.02.2023).

[27] "Push API - Web APIs | MDN". (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Push_API. (accessed 09.02.2023).

[28] "Web Authentication API - Web APIs | MDN". (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API. (accessed 09.02.2023).

[29] D. Taft, "Ionic Capacitor emerging as successor to Cordova", 2021. [Online]. Available: https://www.techtarget.com/searchmobilecomputing/news/252496257/Ionic-Capacitor-emerging-as-successor-to-Cordova.

[30] "Add to Home screen (A2HS) | Can i use ... Support tables for HTML5, CSS3, etc". (2023), [Online]. Available: https://caniuse.com/web-app-manifest. (accessed 10.02.2023).

[31] J. Simmons, "News from WWDC22: WebKit Features in Safari 16 Beta", 2022. [Online]. Available: https://webkit.org/blog/12824/news-from-wwdc-webkit-features-in-safari-16-beta/#web-push-for-macos.

[32] O. Hoang, "Why should you publish your PWA on app stores and how to implement it?", [Online]. Available: https://blog.arrowhitech.com/publish-pwa-on-app-stores/.

[33] M. Tatis, "Can You Put a PWA on the App Store?", 2021. [Online]. Available: https://www.koombea.com/blog/can-you-put-a-pwa-on-the-app-store/.

[34] "App Store Review Guidelines - Apple Developer". (2022), [Online]. Available: https://developer.apple.com/app-store/review/guidelines/. (accessed 10.02.2023).

[35] "Writing Markup With JSX - React". (2023), [Online]. Available: https://react.dev/learn/writing-markup-with-jsx. (accessed 24.03.2023).

[36] "Core Components and Native Components · React Native". (2023), [Online]. Available: https://reactnative.dev/docs/intro-react-native-components. (accessed 24.03.2023).

[37] "Style · React Native". (2023), [Online]. Available: https://reactnative.dev/docs/style. (accessed 24.03.2023).

[38] "Forms - React". (2023), [Online]. Available: https://legacy.reactjs.org/docs/forms.html. (accessed 24.03.2023).

[39] C. Simmons, "Ionic vs. React Native: Performance Comparison", 2022. [Online]. Available: https://ionic.io/blog/ionic-vs-react-native-performance-comparison.

[40] D. Rykun, T. Malbranche, N. Corti, and L. Sciandra, "Announcing React Native 0.70", 2022. [Online]. Available: https://reactnative.dev/blog/2022/09/05/version-070.

[41] "About Android App Bundles | Android Developers". (2023), [Online]. Available: https://developer.android.com/guide/app-bundle#size_restrictions. (accessed 12.04.2023).

Appendix A Mobile device details

Device name	Honor Play
Model	COR-L29
Build number	9.1.0.406
Android version	9
CPU	HiSilicon Kirin 970
RAM	4,0 GB
Total internal storage	64,00 GB
Screen resolution	2340 x 1080